A Locally Nameless Theory of Objects

L. Henrio, F. Kammüller, B. Lutz and H. Sudhof
CNRS – I3S – INRIA, Sophia-Antipolis and Technische Universität Berlin

I. Introduction

This paper presents the formalisation of an object calculus in Isabelle/HOL highlighting the binder technique called locally nameless[1]. This techniques has its origins already in a note at the end of de Bruijn's paper [5] introducing the classical de Bruijn indices. In the last few years, with the advent of mechanized proofs in the domain of programming languages, e.g. [1], this technique attracted new attention. The most recent work on locally nameless technique [2] provides cofinite quantification, necessary for proving non-trivial properties. Indeed the de Bruijn indices are often criticised, as being too technical, that is why alternative techniques are investigated. The de Bruijn indices method, however, is known to be reliable, and is often chosen in order to focus on aspects of programming languages unrelated to variable bindings. With locally nameless techniques, one expects to spend less time proving auxiliary lemmas dealing with variable bindings, but also to obtain theorems that are more convincing because closer to the paper version. Our contributions are a formalisation in Isabelle/HOL of ς-calculus; and an in depth comparison of both locally nameless and de Bruijn complete mechanisations including specification and proofs.

II. Binder Techniques

The formalisation of programming languages in rigorous frameworks has revealed some crucial issues summarised in the POPL-mark challenge [1], where the representation of binders is a central problem. Intuitively, a language that has local scopes and parametrisation – for example functions $\lambda x.fx$ – needs to refer to the formal parameters – here x – when they occur inside these scopes. The natural, human understandable way is to use variables, like x, to define and denote formal parameters by name, but variables are not well suited for mechanisations. Variable capture may occur: a free variable x in a term t may accidentally be "captured" when substituting t inside a scope where x bound. To avoid this, we use a consistent renaming, α-equivalence (renaming of bound variables). However, α-equivalence creates equivalence classes making equality and proofs of theorems harder to handle.

De Bruijn Indices: The solution proposed by N. G. de Bruijn, is to replace each occurrence of a variable by an integer equal to the number of binders that have to be crossed to reach the binder for the considered variable: a variable is replaced by the distance from its binding scope. For example, the λ-term $\lambda x.x(\lambda y.x\,y)$ becomes $\lambda(0(\lambda 1\,0))$. Unfortunately, substitution becomes technical because of the "lifting" of indices when entering a binder, or replacing a term under binders.

This work was supported by DFG project Ascot and Aspen (grants Ja 379/18-1 and Ka 2757/1-1).

[1]http://afp.sourceforge.net/entries/Locally-Nameless-Sigma.shtml

Locally Nameless: The de Bruijn method can be refined in order to avoid manipulation of explicit indices. For this, the principle of locally nameless representation is to use indices to represent bound variables, and classical named variables to represent free (unbound) variables. Open and close operations translate between those representations [2]. This technique is attractive as it combines unique representation, with human understandable expression of specification.

The *open* operation, written t^u, substitutes a term u for the outermost bound variable, in the term t. For example $(bvar\,0\,\lambda((bvar\,1)(bvar\,0)))^n$ is equal to $n\,\lambda(n\,(bvar\,0)))$. The opposite operation *closes* a term: given a name, the closing replaces the occurrence of variables of this name with an index for a variable bound at the outermost level.

In the locally nameless approach we must use only well-formed terms, where *bound* variables are represented by indices. The notion of locally closed terms ensures this e.g. $\lambda(bvar\ 2)$ is not locally closed. Local closure of terms is a necessary requirement for most theorems. Another problem arises when reducing a term under a binder. Useful properties should be valid when closing a term under *any fresh variable*. We need: $\forall x \in FV(t).t^x \rightarrow (t')^x \Longrightarrow \lambda(t) \rightarrow \lambda(t')$. The drawback of this proposition is that it is sensitive to the set of free variables, that may vary in an unexpected way. The approach of *cofinite quantification* [2] should be used: we abstract over the set of free variables $FV(t)$, and let fresh variable range over the complementary of an *existentially quantified finite* set L:$\exists L$ finite.$\forall x \notin L \ldots$. This set can then be instantiated appropriately, when handling proofs.

Nominal Techniques: Another approach, proposed by Urban based on Pitts' work on nominal logic [9], is called nominal technique [11]. Here, terms are identified as a set bijective to all terms factorised by α-equivalence. The classical hypothesis, "there is a fresh variable" for a term t is replaced by "there is a finite *support* for x": the set of atoms used in t is finite, and infinitely many "fresh" atoms are available. Unfortunately, we cannot use the Isabelle/HOL package for nominal techniques as it is, because our terms contain finite maps, and while it is trivial that a finite maps guarantee finite support, such a reasoning is not yet supported by Urban's package.

Higher Order Abstract Syntax: In HOAS binders are directly represented by binders of the meta-level [10]. The encoding is more direct than in the other approaches, but HOAS is restricted when it comes to meta-level reasoning [7].

III. Locally Nameless ς-Calculus

Objects are in our formalisation defined as sets of unordered labelled methods $[l_i = \varsigma(x,y)b]^{i\in 1..n}$ (object "fields" are considered as methods with no parameters). ς binds two

variables: the self x and a method parameter y in each method. The object language features method call $t.l(s)$, and method update $t.l := \varsigma(x,y)b$ on objects. This calculus corresponds to the definitions of the ς-calculus of Abadi and Cardelli [3] but introduces an explicit method parameter beside self. An object is a finite map from label names to objects. The term $[l_1 \mapsto [], l_2 \mapsto \varsigma(x,y)x].l_1 := \varsigma(x,y)y$ defines a simple object, and modifies one of its fields.

A. Syntax

The syntax of objects with locally nameless representation differs not very much from the de Bruijn representation. Indeed, the only difference is the additional constructor `Fvar` introducing named `fVariables` as terms. Those are for convenience chosen to be the type `string`.

```
datatype bVariable = Self nat | Param nat
types fVariable = string
datatype term =
    Bvar bVariable
  | Fvar fVariable
  | Obj (Label ⇒f term) type
  | Call term Label term
  | Upd  term Label term
```

The datatype `bVariable` comprises two *exclusive* possibilities (a datatype is a generalized sum type): it can represent the self parameter, or the added (second) method parameter. Here is the example object shown above.

```
Upd (Obj [l1 ↦ empty, l2 ↦ Bvar (Self 0)] T)  l1 (Bvar (Param 0))
```

Opening a term at a bound variable corresponds to a kind of instantiation of this bound variable with a given subterm. Thus, opening can also be seen as a form of substitution. We will in fact directly employ opening as a substitution when defining the semantics. Hence, the following definition's core part is the first clause, the others just pass the recursion into the term structure. This first clause replaces a bound variable if `n` matches the index of the parameter. Due to the two parameter types of our terms, we always open with a pair of terms and replace, depending on whether the bound is `Self` or `Param`, by the first or second element of the pair, respectively.

```
open :: [nat, term, term, term] ⇒ term ("{_ → [_,_]} _")
and open_option :: [nat, term, term, term option] ⇒ term option
where
 op_Bvar:{k→[s,p]}(Bvar b) =
      (case b of (Self i) ⇒ (if (k = i) then s else (Bvar b))
               | (Param i) ⇒ (if (k = i) then p else (Bvar b)))
...
|op_Obj :{k →[s,p]}(Obj f T)=Obj(λl.open_option(Suc k) s p (f l)) T
|op_None:open_option k s p None = None
|op_Some:open_option k s p (Some t) = Some ({k→[s,p]}t)
```

Let us only describe the most characteristic case: `open_Obj`. Recursive opening inside the object is defined by mapping a function (`λl. ...`) on all its methods (most of them being undefined, `None`). This explains why we use two mutually recursive functions `open` and `open_option`, one of them accepting `Some term` or `None`. The function applied to each member method is the recursive application of open, but with `Suc k` as index because we entered a binder (similarly to what we would do for de Bruijn method).

To abstract a variable, `close` is defined similarly. As close corresponds to a method abstraction we chose the syntax `{_ ←[_,_]} _`. The meaning of close as an abstraction is reflected by the choice of the syntax for the operator for closing at `0`.

```
ς [s,p] t == {0 ← [s,p]}t
```

Opening and closing efficiently convert free and bound variables backwards and forwards. Remember that the coexistence of free and bound variables necessitates restricting propositions to manipulate only terms without "unbound bound variables", i.e. *locally closed terms* (written `lc t`).

B. Cofinite Quantification

One problem when changing between free and bound variables is the need for fresh variables. Otherwise, two originally different variables might be considered as the same one. Hence, whenever we have a rules which uses a newly introduced variable name, we need to find a fresh name. Technically, we can use a function `FV` collecting the free variables of a term, and add the additional premise `x` $\notin$ `FV(t)` whenever a fresh variable name x is required. This way of formalising can be described as the "exists-fresh" approach [2]. For example, suppose that `t` is a subterm under a binder, to make it locally closed, we need to instantiate the top-level bound variable of `t`: $\mathtt{t}^{[\mathtt{s},\mathtt{p}]}$, but to keep the original term `t` (and open the term later with `s` and `p`), we need `s` and `p` fresh.

The "exists-fresh" approach leads to very clumsy proofs: intuitively, we need to prove statements for a set of free variables differing from the ones given as hypotheses. In recent work by Aydemir *et al.* [2], a more sophisticated technique called cofinite quantification is introduced that eases the proofs involving such rules. The basic idea (cf Section II) is to abstract from sets of free variables $FV(t)$, but instead consider some arbitrary finite set L, i.e. assuming a "cofinite set" of variable names. Since L is arbitrary, it can be chosen later as a convenient set bigger than the set of free variables. Any naïve way using simply locally nameless representation *without* using cofinite induction in the semantic definition would lead to unsolvable proof obligations for some theorems. Thus the semantics of our calculus is expressed by rules of the form:

$$\frac{\text{finite } L \qquad \mathtt{lc}\ o \qquad \forall x\ y.\ x{\neq}y \wedge x,y{\notin} L \Longrightarrow \exists t''.(t^{[\mathtt{x},\ \mathtt{y}]} \rightarrow_\varsigma t'' \wedge t' = \varsigma[\mathtt{x},\ \mathtt{y}]t'')}{o.l := t \rightarrow_\varsigma o.l := t'}$$

C. Proved Properties

A basic property of our object theory is confluence.

Theorem 1: $t \rightarrow_\varsigma t_0 \wedge t \rightarrow_\varsigma t_1 \Rightarrow \exists t'.t_0 \rightarrow_\varsigma^* t' \wedge t_1 \rightarrow_\varsigma^* t'$

In an initial experiment we had first mechanized this proof for a simplified ς-calculus with lists and de Bruijn technique [6]. We adapted this proof to locally nameless representation.

We also specified a type system for the calculus and proved *preservation* and *progress*.

IV. Comparison with de Bruijn technique

The first crucial point of comparison between the different binder representations is the point of view of the formalisation.

Two criteria are important here: how easy it is to write the formalisation, and how easy and convincing it is to read it.

The advantage of the locally nameless formulation is the closeness to paper style notation. In the specification of the syntax and semantics we often encounter some technical overhead due to the new constructors for free variables. Moreover, we need to establish the well-formedness of terms by adding *lc* predicates to the premises of the reduction rules. Fortunately, the additional *lc* condition mainly states that substituted terms correspond to correct ς-calculus terms.

Let us focus on the reduction inside binders. Specifying that any field can be reduced in de Bruijn notation leads to the rule:

```
Obj: ⟦s →ς t; l ∈ dom f⟧⟹ Obj (f (l ↦ s)) T→ςObj (f (l ↦ t)) T
```

This is very similar to the paper version. The locally nameless is less straightforward: we need cofinite quantification:

```
Obj: ⟦ l ∈ dom f; finite L; ∀l∈dom f. body (the (f l));
∀s p. s∉L ∧ p∉L ∧ s≠p⟶
     ∃t''. t^[Fvar s,Fvar p] →ς t'' ∧ t' = ς[s,p] t'') ⟧
      ⟹  Obj (f(l ↦ t)) T →ς Obj (f(l ↦ t')) T
```

Additional requirements refine what is meant by "reduce under the binder"; in fact the difficulty is to make the sub-term under the binder locally closed before reducing it, which somehow refines the intuitive notion of (correct) reduction under binders.

The essential relations of the calculus, reduction and typing, are not more readable in their respective locally nameless versions, compared to their de Bruijn incarnations. In both formalisations, the introduction of syntactic sugar can bring some rules very close to a paper version. Some advantages are apparent: the more restrictive reduction relation for locally nameless variables is closer to the version found on paper, as it does not apply to terms with dangling indices.

Concerning proofs, the notable benefit comes from the explicit distinction between the variable types, which can improve readability and ease reasoning for many lemmata, especially the basic lemmata and confluence proofs.

Several lemmata about various translations between the two kinds of variables – bound and free – used in the locally nameless version require rather technical proofs. The technicality of the proofs stems to a large part from the more complex induction schemes, caused by the more complex reduction relation. By contrast, the de Bruijn version's reduction relation and thus induction scheme is much easier to use. This advantage, however, is negated by the lemmata required for the *lifting* in conjunction with the auxiliary constructs which are of comparable complexity and arguably even less readable.

Concerning typing, the locally nameless formalisation improves the understandability of proofs, but at the price of rather technical lemmata for renaming. We are not able to observe a major improvement in the complexity of the major proofs, but for the most part, there is no notable burden either. The proof principles are similar for either variable representation.

V. Conclusions

The clear advantage of the locally nameless formalisation is the handling of free variables. The de Bruijn version did not allow reasoning about free variables for a very simple reason: it is not possible to express free variables. More precisely, unbound de Bruijn indices could sometimes simulate free variables, but such a solution is unsatisfactory because the intent of a free variable is different from a dangling index. Moreover, the explicit distinction between bound and free variables eases the handling of either kind of variable and enhances the readability of proofs and formalisations. Cofinite quantification, freshness and renaming are the major reasons for additional and technical proofs in the locally nameless representation, and all of these items are required for the reasoning about named free variables. The locally nameless rules are more complex than their de Bruijn counterparts because the locally nameless representation introduces new concepts and is precise about well-formedness and closure. This initial formal overhead is paid back by a natural notation in theorems, and by improvement for interactive proofs.

This paper also shows that locally nameless techniques can be easily adapted to the modelling of an object language, and to the use of binders for multiple parameters.

References

[1] B. Aydemir, A. Bohannon, N. Foster, B. Pierce, J. Vaughan, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic, M. Fairbairn, P. Sewell. *The POPLmark Challenge.* `http://alliance.seas.upenn.edu/ plclub/cgi-bin/poplmark/`.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, S. Weirich. Engineering Formal Metatheory. *Principles of Programming Languages, POPL'08*, 2008.

[3] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[4] Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, 2nd edition, 1984.

[5] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, **34**, 1972.

[6] L. Henrio and F. Kammüller. A Mechanized Model of the Theory of Objects. *FMOODS 2007*. LNCS **4468**, Springer, 2007.

[7] F. Honsell, M. Miculan, and Ivan Scagnetto. Pi-calculus in (Co)inductive-type theory. *Theoretical Computer Science***253**(2):239–285, 2001.

[8] F. Kammüller and H. Sudhof. Towards a Mechanized Theory of Aspects. em Theorem Proving in Higher Order Logics 2009. Emerging Trends.

[9] A. M. Pitts. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation*, **186**:165–193, 2003.

[10] C. Roeckl and D. Hirschkoff. A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *Journal of Functional Programming*, **13**:415–451, 2003.

[11] Christian Urban et al. Nominal Methods Group. Web-page at `http://www4.in.tum.de/~urbanc/Nominal/`.